Report No. 313

SOFTWARE DEVELOPMENT FOR THE ARRAY COMPUTER ILLIAC IV

by

Robert S. Northcote

March 12, 1969

ILLIAC IV Document No. 214

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

Report No. 313

SOFTWARE DEVELOPMENT FOR THE ARRAY COMPUTER ILLIAC IV

by

Robert S. Northcote

March 12, 1969

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

* This work was supported in part by the Advanced Research Projects Agency as administered by the Rome Air Development Center under Contract No. US AF 30(602) 4144 and in part by the Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, March 12, 1969.

ABSTRACT

This paper is a survey of the software development for ILLIAC IV. A brief description of the ILLIAC IV hardware system and its memory organization is given. The software is being implemented on a Burroughs B5500 for eventual use on a B6500. The structure of the operating system and the functions of the operating system modules are described to illustrate the somewhat different approach necessary in processing ILLIAC IV programs. The translator writing system, which provides for the automatic generation of syntax directed recognizers and facilitates the construction of other parts of translators, is described. Multipass translators with a simple structure can be obtained from the system on suitable specification of the syntax and semantics of the source language. A brief description of TRANQUIL, an algorithmic language for array processing, is given. Some of the techniques for handling large arrays and mapping functions to specify storage methods for them are mentioned.

INTRODUCTION

The array computer ILLIAC IV and its organization with an associated B6500 computer have been described by Barnes, Brown, Kato, Kuck, Slotnick and Stokes (1968). The system is designed to have an array of 256 coupled processing elements (PE's) arranged in four quadrants in each of which 64 PE's will be driven by decoded instruction signals emanating from a single control unit (CU).

The PE's of a quadrant must simultaneously carry out the same operation on the operands to which they each have access and thus will operate in parallel. Each of the 256 PE's will have an instruction set which includes floating point arithmetic on both 64 bit and 32 bit operands with options for rounding and normalization, 8 bit byte operations, and a wide range of tests due to the use of addressable registers and a full set of comparisons. The PE's differ from conventional computers in three main ways. Firstly, each is capable of communicating data to its four neighboring PE's in the array by means of routing instructions. Secondly, each PE is able to set its own mode registers thus enabling or disabling itself for the transmission of data or the execution of instructions from its CU. Thirdly, all instruction decoding and some other functions are effected in the CU's, thus eliminating the need for a lot of hardware in the PE's. Each CU contains a 64 word instruction buffer, a 64 word local data buffer and 4 accumulator registers (CAR's) which are loaded, as required, from memory.

The four quadrants (each with its 64 PE's and 1 CU) may be operated independently, in pairs, or all together. This allows some flexibility in the fitting of problems to the system. There is no other provision for multiprogramming or multiprocessing. In the united configuration all 256 PE's are effectively driven by 1 CU and the system will be capable of executing of the order of 10^9 operations per second.

Memory Usage and the B6500

Each of the PE's will have 2048 (or more) words of 64 bit semiconductor memory with a 200 nanosecond cycle time. Thus the primary memory may be regarded as an array of 2048 rows (words) by 256 columns (PE's), or as 4 arrays of size 2048×64 as illustrated in Figure 1. The secondary memory will be a parallel head per track disk arranged with nine storage units linked to each of two electronics units. Total capacity will be 1.0×10^9 bits with a transfer rate to or from the primary memory of $.5 \times 10^9$ bits per second from each electronics unit for an effective transfer rate of 10^9 bits per second. Thus, the transfer of a 256×256 array between primary and secondary memory (through one electronics unit) will take 8 milliseconds. The necessity of having such a fast I/O channel is more obvious when it is noted that multiplication of two 256×256 arrays will take about 80 milliseconds.

Such speeds of operation and data transfer imply that there should not, in general, be any transfer of data between the system described above and tertiary storage (other peripheral devices) once

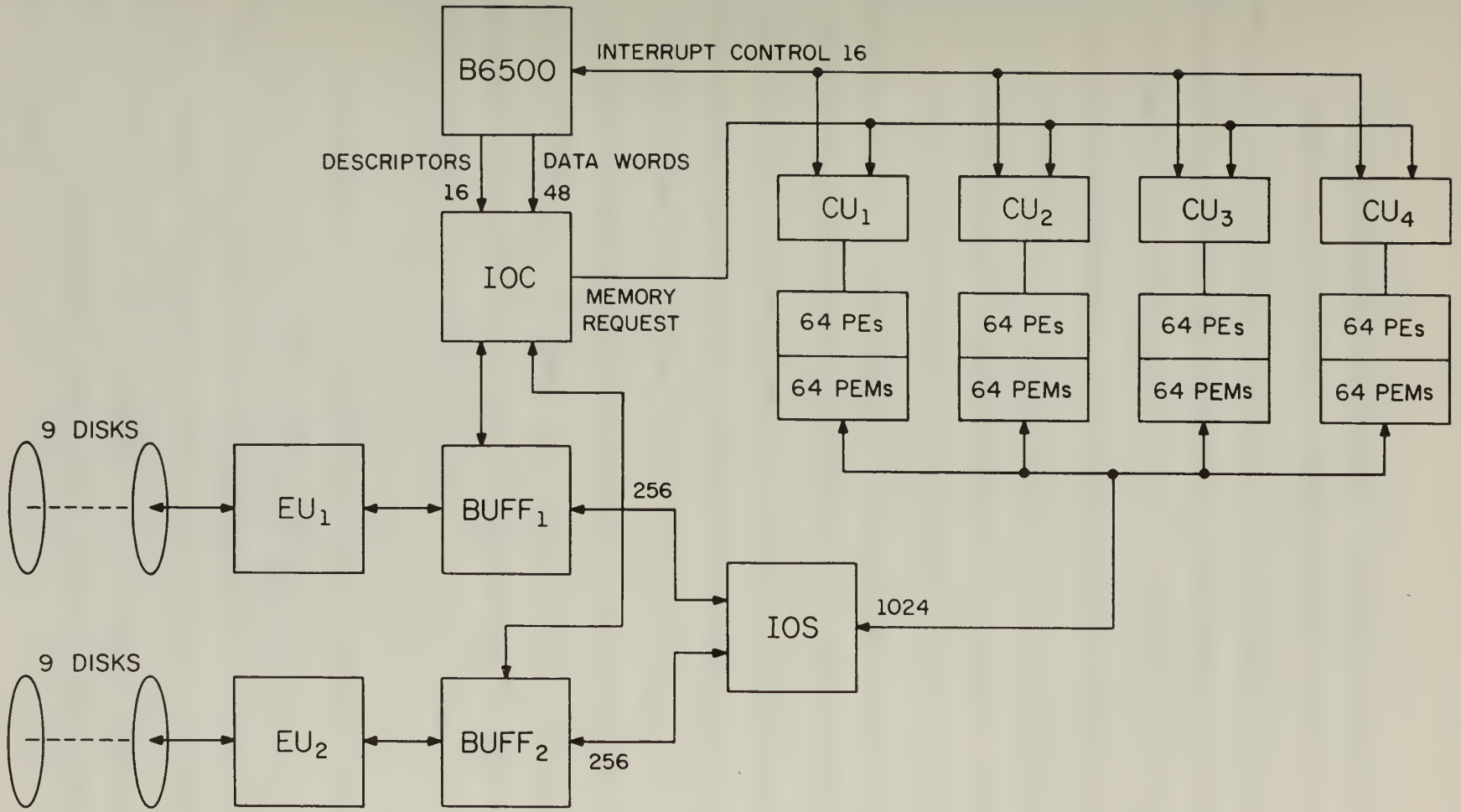


Figure 1. The organization of the ILLIAC IV system.

a program has begun execution. Hence all executable code and data required by a program must be loaded into primary and secondary memory before the program is initiated. The supervision of this task, together with the translation of source programs, the configuring of the quadrants for different jobs, and the manipulation of files for both pre and postprocessing are essentially nonparallel operations which are performed in the final main component of the system, a Burroughs B6500 computer. The complete system configuration is illustrated in Figure 1. Almost all the software for the entire system is controlled and executed in the B6500. All data in tertiary memory must pass through the B6500 to a 4K word PE-type memory buffer before being loaded into primary or secondary memory. The buffer is contained in the input/output controller (IOC) which also contains a 24 word queuer of descriptors for data transfers between the buffer (linked to tertiary memory), the disk (secondary memory) and PE (primary) memory. The descriptors are loaded from the B6500. Hence an executing ILLIAC IV program which requires a transfer between primary and secondary memory must first cause an interrupt of the B6500 before a descriptor can be loaded in the IOC and the transfer can begin. To minimize the frequency of these interrupts it is possible for the B6500 to ready an arbitrarily long list of descriptors on a single interrupt. The only software resident in the ILLIAC IV itself will be a small part of the operating system, to allow for the processing of traps and the signalling of I/O requests, and parts of the loader, because relocation can be done efficiently on ILLIAC IV.

The design and construction of the hardware and software for the complete ILLIAC IV system is a joint effort of the University of Illinois and Burroughs Corporation. The latter is responsible for the detailed design and actual construction of the hardware, while the software is the sole responsibility of the University. Since almost all the software is to be run on a B6500, software development has been able to proceed apace on a B5500, the hardware and software design of which are very close to that of the new B6500. Detailed software design was initiated in the first quarter of 1967 and implementation was begun, writing almost exclusively in Burroughs Extended Algol, in the final quarter of 1967 when a B5500 was delivered to the university. Although ILLIAC IV design automation, diagnostics and simulated applications programs are now using a large proportion of the B5500 facility the almost exclusive use of the B5500 in the early stages and the use of a highly flexible higher level language have been major factors in the fairly satisfactory progress of software development with minimal personnel resources. Some of the early software design had been described by Kuck (1968).

THE OPERATING SYSTEM

Access to the ILLIAC IV may only be obtained through the B6500 to which entry may be made via a batch process, via one of several local terminals, or via one of two networks (the nationwide ARPA contractors net or the University of Illinois ILLINET). In addition to satisfying these users the B6500 must process interrupts from the ILLIAC IV (its highest priority user) on termination of a

job or a request for data transfer. The B6500 resident ILLIAC IV operating system (OS) must supervise all these tasks if it is to give high priority to ILLIAC IV interrupts. Most of the tasks processed, such as file editing, the pre and postprocessing of data for ILLIAC IV jobs and the translation of source programs for the ILLIAC IV will, however, be controlled by the standard B6500 master control program (MCP) once they are initiated by the OS. To facilitate an understanding of the operating system structure a discussion of the steps in processing an ILLIAC IV job is now given.

To run a job on the ILLIAC IV a user will submit a source program for translation (a B6500 task) and the object code will be stored as a file on a B6500 peripheral. The job can be scheduled for the ILLIAC IV only after all input data have been loaded into the secondary memory. Some source programs, particularly those written in TRANQUIL (the algorithmic language for ILLIAC IV programming) specify specialized mapping functions which must be applied to data before they are loaded into secondary memory. Thus, two specifications of input (and output) data will normally exist; one provided by the user to specify how the data are stored (and their format) in tertiary storage (B6500 disks, tapes, cards, etc.) and the other provided by the TRANQUIL compiler (or the user) to specify how the data are stored in primary and secondary memory. Only when both the specifications and the data are available can the OS schedule the B6500 to perform the preprocessing and loading of the data, provided sufficient secondary memory is available. After loading of data has been completed, the OS

can schedule one or more ILLIAC IV quadrants into which the object code will be loaded and executed. During processing the ILLIAC IV will (normally) interrupt the B6500 to request data transfers between primary and secondary memory, and will also cause an interrupt on completion of the job. The OS must then schedule the unloading of the users data from secondary memory to tertiary storage. The user then may postprocess and/or view all or part of his output files via displays, plotters or some high-speed hard copy generator.

Jones (1968) has given a careful analysis of the basic requirements for a good operating system. He has expressed the conviction that:

(i) an operating system structure should be defined by a minimum system--"the lower system limit of a modular open-ended dynamic growth system;

(ii) the supervision of all tasks should be through the manipulation of a collection of files and tables;

(iii) it should be possible to monitor the performance of the system and modify the scheduling algorithms via a display console.

The ILLIAC IV operating system is being designed to meet these criteria. Each of the services alluded to in the paragraph above will be implemented by one or more program modules within the system. An illustration of these modules and the paths between them is given in Figure 2.

The user interface to the OS is through the job parser which can be activated by any available B6500 input media such as card readers, tape drives and user consoles, or by other running B6500 programs which

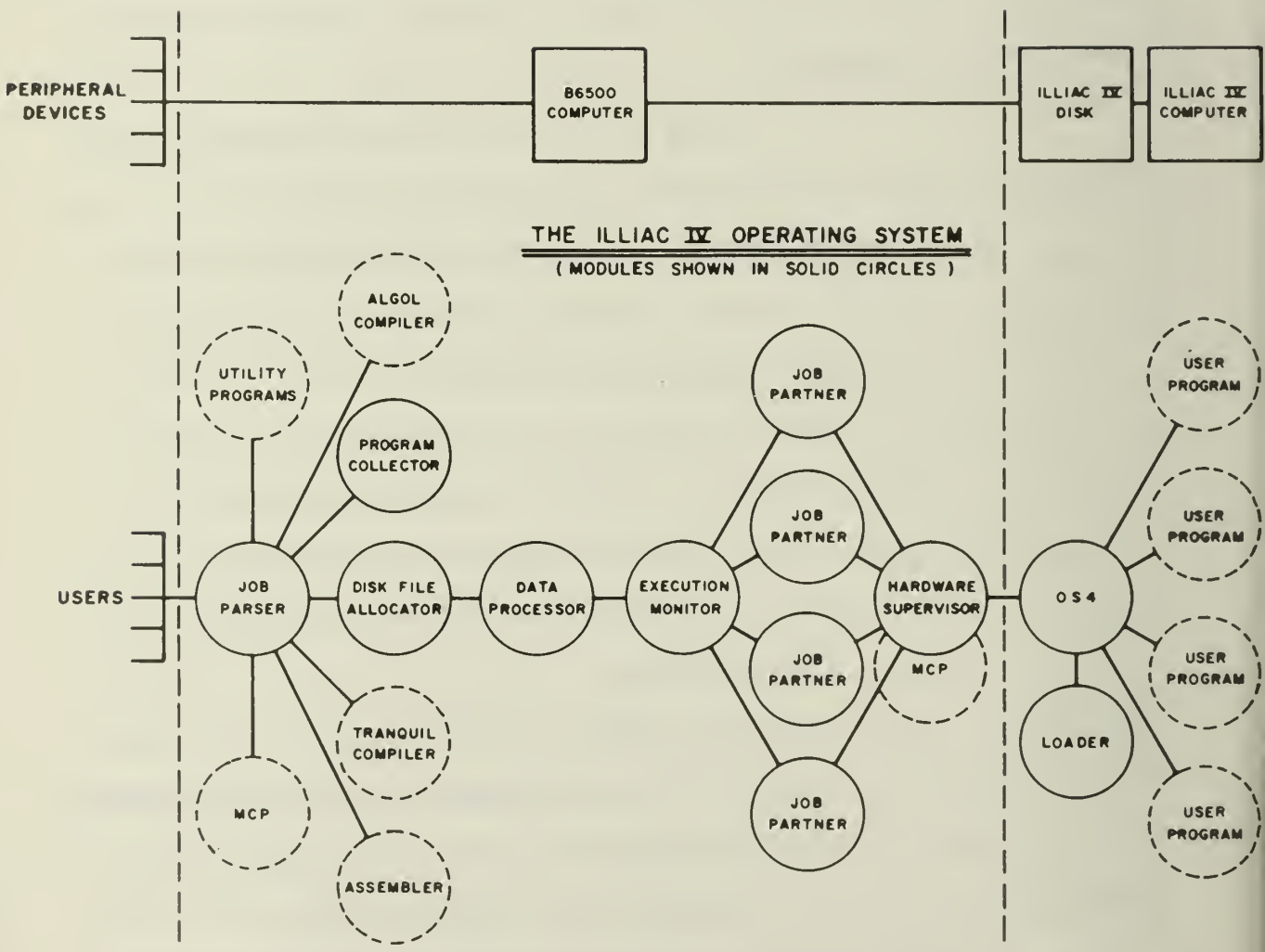


Figure 2. The structure of the ILLIAC IV operating system, with its modules shown by solid circles.

need to initiate new jobs. The parser scans the ILLIAC Control Language (ICL) statements of a job and builds a job dossier before providing interfaces to a variety of utility programs and to other parts of the operating system. The utility programs, some of which will be conversational for console users, perform file editing, program segment collection, data pre and postprocessing, and translation functions. Virtually any utility program can be added; for example, the job parser will have a conversational interface to an education utility program that will teach users how to use the system and will provide documentation. ICL provides external specification of job steps and facilitates the passing of parameters to these job steps by the user. The execution of a utility program by the B6500 requires the specification of suitable control information to be used by the MCP in addition to the ICL statements which are needed to gain access to the system and the required program(s). The job parser will also interface with the operator's console through which the operator will be able to monitor the state of the system and modify the system parameters.

The program collector carries out various operations on program segments to prepare complete programs for execution on ILLIAC IV. These tasks include reformatting segments output by translators into a form suitable for input to the loader, gathering together program segments from various files, arranging the structure and mapping of overlayable programs, and constructing and maintaining system and user program libraries.

When the job parser is satisfied that all translating and data reformatting by utility programs, and segment binding by the program

collector have been completed for a job, the job is ready to be scheduled for the ILLIAC IV. The scheduling of ILLIAC IV resources is performed by the disk file allocator, data preprocessor and execution monitor modules. It is initiated when the job parser passes a job dossier pointer value to the disk file allocator. These three modules queue requests for assignment and use of buffer memory, secondary storage, and ILLIAC IV quadrants. They also move files between tertiary and secondary storage before and after job execution. When each module finishes processing a job, it is sent on to the next module for processing by passing a pointer to the job's dossier along the paths shown in Figure 2.

The execution monitor initiates and terminates ILLIAC IV execution, regulates the use of the buffer memory and queues requests from the data processor. A job is initiated by starting a B6500 job partner program which communicates with the hardware supervisor module, the execution monitor and the MCP. It initiates ILLIAC IV processing, processes ILLIAC IV data transfer requests, answers other communcations including error interrupts and issues commands to OS4 -the ILLIAC IV resident operating system module. The hardware supervisor module is actually a group of ILLIAC IV input/output procedures which are embedded in the MCP from where they are accessible by all job partners. The execution monitor provides tables for the job partners, including a file map table which is used by the hardware supervisor procedures to build data transfer descriptors. If a multiquadrant job decides to split, its job partner may request the execution monitor to initiate new job partners for the appropriate quadrants and the execution monitor will initiate them. An

ILLIAC IV job is terminated when all its job partners have been terminated. The execution monitor then passes the job dossier pointer back to the data processor which unloads the secondary memory. Standard job partner programs will be provided but users may write their own. For this reason, job partners will not be able to modify entries in the file map and other important tables.

The real bottlenecks likely to be encountered by the operating system will arise from the multiquadrant use of ILLIAC IV and from the limited amount of secondary memory. Since the disk system is only about 40 times the size of the primary memory and some jobs require several primary memory loads, there is not enough space to ensure the establishment of a large queue of ready-to-run jobs on the disk. Also, special patterns of data may be required on the disk for some jobs. It appears improbable that loads can be leveled sufficiently well to absorb statistical fluctuations when 30-40 small jobs appear to be the maximum population possible and one small one-quadrant job can require checkerboarding of 10 per cent of the disk badly enough to ensure that few other jobs can be in the secondary memory disk queue. It will be desirable, therefore, to keep a few small-disk-requirement, one quadrant jobs available to take up slack if it appears that some quadrants will stand idle for a few minutes.

THE TRANSLATOR WRITING SYSTEM

Many different translators (the term includes compilers and assemblers), ranging from the extremely complex to the rather trivial,

are required for software support of the ILLIAC IV system. Translators have even been constructed for use in the development phase of the project in the design automation and diagnostics areas. There was, therefore, considerable incentive to build a translator writing system (TWS), or compiler-compiler, which would facilitate the implementation of translators. The advantages gained through the more optimum use of limited personnel resources are deemed to outweigh the disadvantages of having slower translators. Another very real advantage is that it should be much simpler to maintain translators which are structurally similar. The use of the TWS will also simplify modification of a translator when the specification of its source language is altered.

The structure of the system is illustrated in Figure 3. It is similar to the systems described by Feldman (1966) and Northcote (1966). However, the system described here is considerably more flexible in the following respects:

(i) the syntax of the source language $\mathfrak{L}$ may be specified in Backus Naur form (BNF), or a modification of it called Translatable Backus Naur form (TBNF) developed by Trout (1969), instead of the more difficult to use Floyd Production Language (FPL);

(ii) the semantics of $\mathfrak{L}$ are specified in Burroughs Extended Algol augmented by the Illinois Semantics Language (ISL) which has some similarity to the Feldman semantic language (FSL);

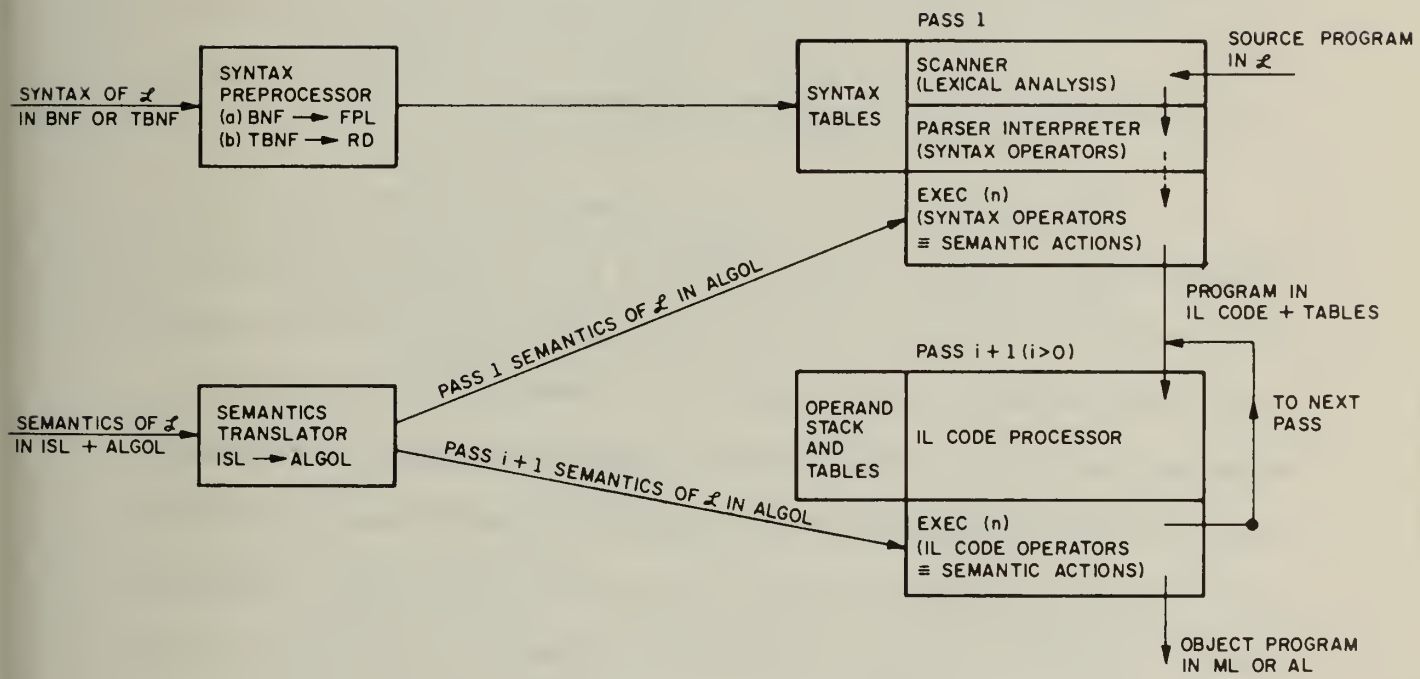


Figure 3. The structure of the translator writing system.

(iii) the operators of the intermediate language (IL) code output by any pass of the translator are specified by the translator writer in the syntax specification, in the CODE generation statements of the semantics and by the provision of a semantic routine in pass i+1 for each operator generated at pass i (the syntax specification is defined as pass 0);

(iv) the definition of IL code permits the specification of an arbitrary number of passes in a translator, in any of which optimization may take place.

The parser for the £ translator may be produced in several different ways. Either a BNF or a TBNF (a brief description is given in Appendix 1) syntax definition may be automatically converted into either a FPL bottom-to-top recognizer or a recursive descent (RD) top-to-bottom recognizer by one of several syntax preprocessors. The recognizers produced may also be either syntax tables which have to be interpreted or may be directly executable Burroughs Extended Algol statements which are plugged into the £ translator. Schemes for obtaining a recursive descent recognizer from BNF (and hence TBNF) are fairly well known, but the generation of a FPL recognizer from BNF, which is briefly described below, is nontrivial. The automatic generation of a recognizer from a language specified by syntax metalanguages such as BNF or TBNF is an added advantage of using a TWS; the recognizer exactly matches a definition which programmers can read (and understand) in a programming manual.

A BNF to FPL translation algorithm has been devised and implemented by Beals (1969) and DeRemer (1969). Consider $\mathfrak{L}$ to be defined by a grammar

$$G = (V_T, V_N, S, P)$$

where

V_T = the set of terminal symbols of $\mathfrak{L}$ (represented by lower case Latin letters);

V_N = a set of nonterminal symbols (represented by upper case letters);

$S \in V_N$ is the objective symbol;

P = a numbered set of BNF production rules defining $\mathfrak{L}$, together with $\Sigma ::= \perp S \perp$.

If α represents the first n symbols on the right of BNF production number π, then the following BNF to FPL mapping rules apply:

	BNF production	FPL statement
(i)	$M ::= \alpha N \ldots$	$\alpha\vert$ *Nh
(ii)	$M ::= \alpha t \ldots$	$\alpha\vert$ *t(π,n+1)
(iii)	$M ::= \alpha$	$\alpha\vert \rightarrow M\vert$ Mt

In these FPL statements the string α is to be compared with the top n symbols in the recognition stack; * denotes scan another symbol from the source program into the stack; $\rightarrow M\vert$ means replace the string α in the stack by the nonterminal symbol M. The symbols on the right are labels

of other FPL statements to be executed; Nh identifies a group of statements which attempts to locate an initial (head) symbol of N in the stack; $t(\pi,n+1)$ labels the statement group which attempts to find a terminal t in the top of the stack corresponding to the t in position n+1 of BNF production number π, Mt labels the group which attempts to match constructs with an M in the top of the stack.

These rather obvious rules and their interpretation form the basis of the algorithm. It is relatively easy to determine what FPL statement labels are required (three rules), and the description of the FPL statements which must be constructed. However, many ordering rules are needed and many statements preclude each other thereby necessitating some expansion of the context in which the syntactic construct under examination is located. This involves the determination of lookahead symbols (source symbols not yet put into the stack, but held in a buffer) and the algorithm becomes quite messy. However, considerable success has been achieved with the algorithm on several languages specified in BNF and TBNF (when first reduced to BNF) and operational recognizers have been obtained. When conversion is successful the BNF generative grammar G_1 and the FPL recognition grammar G_2 of a language £ are equivalent.

The semantics of each pass of a translator are embodied in Algol procedures EXEC(n), (one for each pass) which are output by the semantics translator shown in Figure 3. Each EXEC(n) is one large CASE statement, each substatement of which corresponds to a different semantic action. Calls on EXEC(n) in pass 1 are made after syntactic

constructs are recognized. These calls are placed in the recognizer by the syntax preprocessor on recognition of pass 1 action identifiers embedded in the BNF or TBNF syntax definition. The semantic actions in EXEC(n) at pass i+1 ($i > 0$) are invoked each time an IL code operator output by pass i is recognized. IL code operands are merely pushed into an operand stack.

A translator generated by the TWS is, thus, a collection of Burroughs Extended Algol program segment files which are concatenated together to form a complete program. After compilation by the Algol compiler the translator is ready for debugging. At least six translators have been, or are being, built using the various processor and program modules provided by the translator writing system. A total of three man years of graduate student effort has been expended on the design and implementation of this system. Parts of the system were useable by the language designers after only one man year of development.

TRANQUIL

If ILLIAC IV, or any other parallel array computer, is to be used effectively it is essential that all possible parallelism be detected in those algorithms which are to be executed by that computer. This is difficult, if not impossible, if the algorithms are specified in languages such as FORTRAN and Algol which essentially express all computational processes in terms of serial logic, as required for conventional computers. Since it is also more convenient for the

user to express array-type computation processes in terms of arrays and parallel operations, rather than having to reduce the inherent parallelism to serial computational form, the specification of a new language for array processor computation is clearly necessary. The TRANQUIL language has been designed to achieve both simpler specifications of, and explicit representation of the parallelism in, many algorithms, thus simplifying the programmer's task and maximizing the efficiency of computation on a computer such as ILLIAC IV. A description of a preliminary version of TRANQUIL has been given by Kuck (1968). Abel, Budnick, Kuck, Muraoka, Northcote and Wilhelmson (1969) have given a TBNF syntax specification and a detailed description of TRANQUIL and the implementation of its compiler. A brief discussion of TRANQUIL is included here for completeness of this paper.

An important consideration in designing a language such as TRANQUIL is that the expression of parallelism in the language should be problem oriented rather than machine oriented. This does not, and should not, preclude programmer specification of data structure mapping at run time, but once the storage allocation has been made, the programmer should have to think only in terms of the data structures themselves. Secondly, the means of specifying the parallelism should be such that all potential parallelism can be specified.

The structure of TRANQUIL is based on that of Algol. Many Algol constructs are used, with the addition of further data declarations, standard array operators and revised loop specifications, including the addition of the set concept. Some of the ideas

embodied in TRANQUIL follow similar constructs in other languages; for example, the index sets in MADCAP described by Wells (1964) and the data structures and operators in Iverson's (1962) APL.

The data structures in TRANQUIL are simple variables, arrays and sets with data-type attributes INTEGER, REAL, COMPLEX or BOOLEAN. Certain precision attributes may be specified. Mapping function specifications, which must be included in every array declaration, are one of the most crucial features in TRANQUIL and the use of ILLIAC IV. Arrays must be mapped to optimize data transfers between primary and secondary memory, to minimize unfilled areas of primary memory, which would otherwise be wasted, and to optimize the use of the PE's. If the STRAIGHT mapping function is used all PE's are activated to access a row (if long enough) of an array, but only one PE is activated to access a column. The SKEWED mapping function rotates the i+1st row across i PE's which enables columns as well as rows of an array to be accessed simultaneously. Detailed analysis of a wide variety of suitable applications and algorithms for their solution on ILLIAC IV has led to the development of other interesting mapping functions. These include CHECKER for partial differential equations applications and STREWED (not yet implemented in TRANQUIL) for the storage of the sparse matrices to be used in the linear programming algorithm discussed by Marceau, Lermit, McMillan, Yamamoto and Yardley (1969). Other useful storage schemes almost certainly will be invented. The user who wishes to specify his own mapping function may do so through the use of special TRANQUIL statements provided for that purpose.

ARRAY PARTITIONING

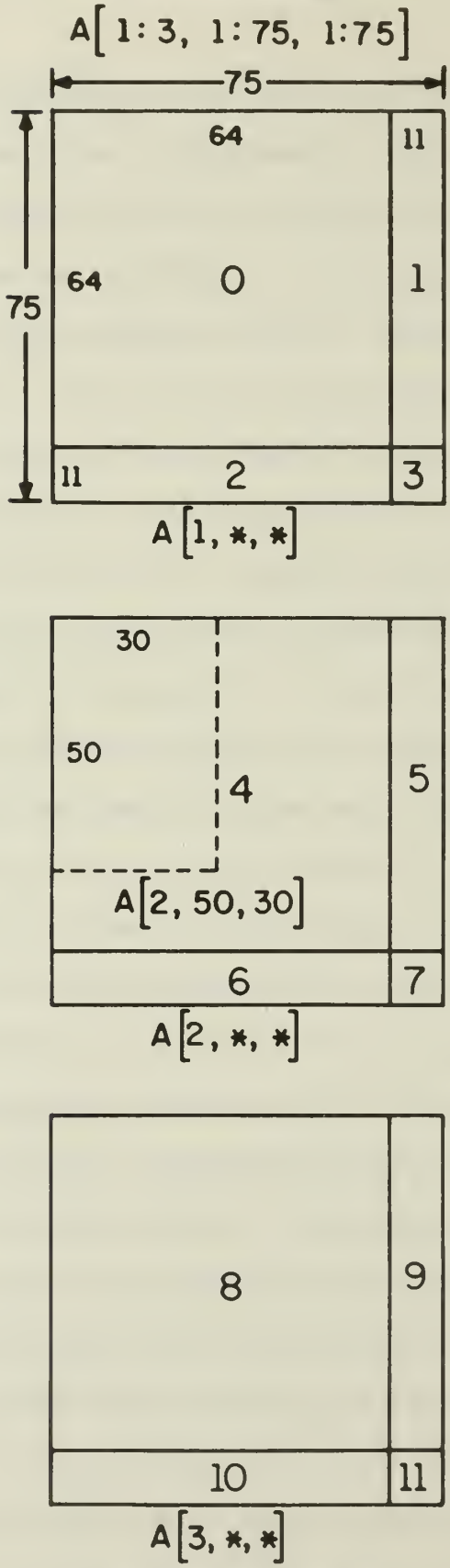
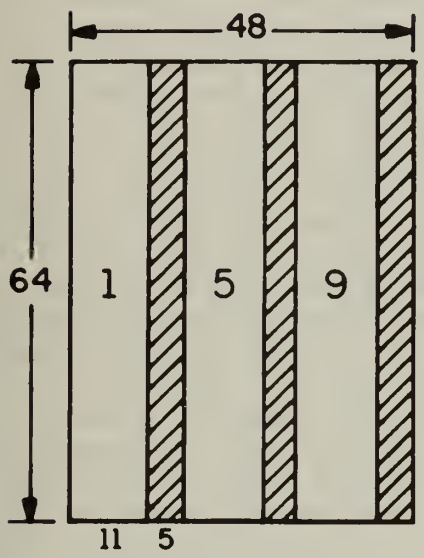


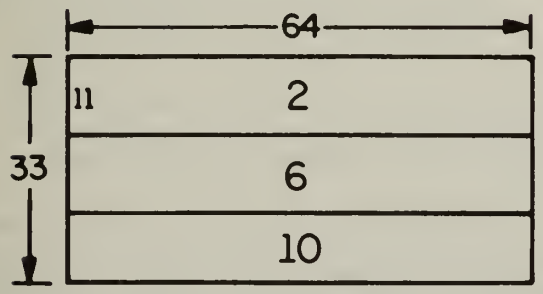
Figure 4. The partitioning of an array A[1:3, 1:75, 1:75] into blocks for array storage.

BLOCK PACKING

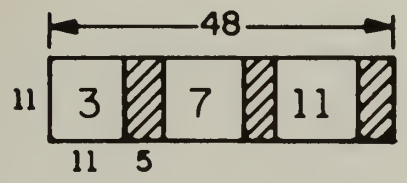
(a)	64 x 64		SQUARE
(b)	m x 64	(m < 64)	HBLOCK
(c)	64 x n	(n < 64)	VBLOCK
(d)	m x n	(m, n < 64)	SBLOCK



PACKING VBLOCKs



PACKING HBLOCKs



PACKING SBLOCKs

Figure 5. Block packing for the array A[1:3, 1:75, 1:75].

The compiler partitions all arrays into 2-dimensional blocks the maximum size of which is $64q \times 64q$ words ($q = 1, 2,$ or 4 according as to whether 1, 2, or 4 quadrants of 64 PE's are to be used, respectively) since primary memory is regarded as an array with 2048 rows $\times$ 64q columns. Figure 4 illustrates the partitioning of a 3-dimensional array into 12 blocks for a one quadrant system. Figure 5 shows how the smaller blocks are packed together to form larger blocks. The partitioning of an array into blocks is usually independent of the mapping function, which is applied after block partitioning. All array operations and data transfers between primary and secondary memory are done in terms of these blocks. Thus blocking facilitates the use of arrays which are larger than primary memory. Blocks are only brought into primary memory when needed. The TRANQUIL compiler automatically generates block transfer requests where they are required in an object program. It will also make some attempt to optimize the use of primary memory through suitable buffering and placement of the requests to account for operating system processing, disk rotation, and loading.

Arithmetic, logical and relational operators and function designators may be used on arrays. The meaning of an operator is determined by the attributes of its operands, which may be simple variables or arrays. All meaningful combinations of operands and attributes are valid; for example, if A and B are arrays then A/B will mean $A \times B^{-1}$ if B has an inverse and the dimensions are correct. The result of a relational operator operating on two array operands is reduced to a

single logical value through use of the qualifiers ANY and ALL; for example:

ANY A < B or ALL A < B

Index sets play an integral part in control statements, where they are used for loop control, for enabling and disabling of PE's, and for PE indexing. A set usually consists of scalar elements but it may also be multidimensional with n-tuple elements ($1 < n < 8$). A set declaration must specify one of the attributes INCSET, MONOSET, GENSET, or PATSET according to whether the set elements are to form an arithmetic progression (increment set), a strictly monotonic set, an arbitrary sequence (general set), or are multidimensional (pattern set), respectively. Several set operations are allowed. Sets with multidimensional elements can be created from sets with scalar or multidimensional elements through the use of the pair (,) and cross product (×) operators; for example:

[1, 2, 3], [5, 10, 15] is [[1, 5], [2, 10], [3, 15]]

[1, 2] × [3, 4] is [[1, 3], [1, 4], [2, 3], [2, 4]]

The operator , has higher precedence than ×.

The major control statements are the SEQ statement (similar to the for statement in Algol) for sequential control, the SIM function and SIM statement for simultaneous control, and a more generalized form of IF statement. If the index sets II and JJ are defined by

II ← [1, 2, 3]; JJ ← [5, 10, 15];

then the following examples illustrate the use of control statements, where I, J, and K are index variables:

(i) FOR (I, J) SEQ (II, JJ) DO B [I, J] ← A [I, J]

is evaluated as

```
B [1, 5] ← A [1, 5];
B [2, 10] ← A [2, 10];
B [3, 15] ← A [3, 15];
```

(ii) FOR (I) SEQ (II) WHILE I < C [I] DO C [I] ← 0

will continue sequential looping until the Boolean expression is FALSE or the index set has been exhausted;

(iii) SIM BEGIN A_1; A_2; . . .; A_n END

causes the assignment statements A_i (i=1, . ., n) to be executed simultaneously using the data that was available before the SIM was encountered;

(iv) FOR (I, J, K) SIM (II × JJ, II) DO D [I, J, K] ← 0

is evaluated as

```
SIM BEGIN D [1, 5, 1] ← 0;
          D [1, 10, 2] ← 0;
          D [1, 15, 3] ← 0;
          D [2, 5, 1] ← 0;
            :
          D [3, 15, 3] ← 0          (9 statements)
    END
```

and the 9 locations are set to zero simultaneously, if possible;

(v) IF FOR (I) SIM (II) ANY C [I] < 7
is an if-clause which has the value TRUE if any one of the first three elements of the array C has a value less than 7.

More sophisticated examples are given in the papers by Kuck (1968) and Abel et al. (1969). A sample program is given in Appendix 2.

Several features of TRANQUIL, notably input/output statements and procedures, have yet to be specified. Most data transfers, between primary and secondary storage, will be implicitly specified in TRANQUIL programs. However, some explicit specification of unformatted data structures will be provided. The provision of additional software to facilitate format-specified transfer of data between tertiary (external peripherals) and secondary storage is planned. As discussed earlier interfaces between such data processing utility programs and the TRANQUIL compiler, which determines the mapping functions, must be provided. Additional features being considered for later incorporation into the compiler include overlayable code segments, quadrant-configuration-independent code, and more specialized data structures and mapping functions.

The task of compiling a language for the ILLIAC IV is more difficult than compiling for conventional machines simply because of the different hardware organization and the need to utilize its parallelism efficiently. Several new compilation algorithms for handling

array mapping functions, index sets, and code optimization have been invented. Completion of the major parts of the TRANQUIL compiler (which utilizes the TWS described earlier) has demonstrated that reasonably efficient object code can be generated for a large number of array-type problems which have been programmed in TRANQUIL. Output from the compiler is currently in ILLIAC IV assembly language to facilitate debugging.

CONCLUSION

Various ILLIAC IV timing and execution simulators and assemblers have also been developed. Early simulator efforts were directed towards determining optimal hardware design characteristics. Since early 1969 an assembler and execution simulator have been used (on the B5500) in the development and testing of many applications programs written in assembly language. Syntactic debugging of TRANQUIL programs has been possible for some time; semantic debugging via the TRANQUIL compiler, assembler and simulator is scheduled for the second quarter 1969. A list processing language GLYPNIR and its compiler (Lawrie, 1969) have also been implemented.

The almost exclusive use of a flexible high-level language such as Burroughs Extended Algol and the ready availability of a multiprogramming computer have been very significant factors in the development of software for the ILLIAC IV system. The early and rapid development of the various packages in the translator writing system has also facilitated software development. These statements are supported by the fact that the software described in this paper has

been developed during the last two years with an average team of two academics, three systems programmers, fifteen half-time graduate students, and a few part-time undergraduates. Much of the software is or will be operational well in advance of delivery of the ILLIAC IV in the latter half of 1970. By that time many applications programs should be thoroughly tested and be ready for full scale production runs.

ACKNOWLEDGEMENTS

The author gratefully acknowledges the assistance of his colleague, Professor David J. Kuck, who initiated and directed most of the work on the TRANQUIL language. Norma E. Abel, P. P. Budnik, Y. Muraoka and R. B. Wilhelmson have dedicated themselves to the design of TRANQUIL and the construction of its compiler. The translator writing system has been enthusiastically developed and implemented by A. J. Beals, J. E. LaFrance, N. C. Machado and H. R. G. Trout. P. A. Alsberg, G. R. Grossman, T. W. Mason and G. A. Westlund are responsible for the design and implementation of the operating system. D. M. Grothe developed the assembly language and its assembler. Grossman was also responsible for the simulator development. Professor Daniel L. Slotnick, director of the ILLIAC IV Project, has provided much inspiration and must be thanked for allowing the software groups to commandeer so many of the brightest and most hard-working graduate students on the Project.

The research and development reported in this paper was supported in part by the Department of Computer Science, University

of Illinois at Urbana-Champaign, and in part by the Advanced Research Projects Agency as administered by the Rome Air Development Center, under contract No. US AF 30(602)4144.

APPENDIX 1: A METALANGUAGE FOR SPECIFYING SYNTAX

Translatable Backus Naur form (TBNF) is specified in a form of BNF which is extended as follows:

1) Kleene star:

< A >* = < > | < A > | < A > < A > | ...

where < > represents the empty symbol

2) Brooker and Morris' question mark (here &):

< A > & = < > | < A >

3) List Facilities

list < A > = < A > < A > *

list < A > separator < B > = < A > [< B > < A >] *

4) Brackets

< T > ::= [< A > | < B > | < C >] < D >

is equivalent to

< T > ::= < R > < D >

< R > ::= < A > | < B > | < C >

5) Metacharacters:

A sharp (#) must precede each of the following characters when they belong to syntactic definitions:

, [,], *, ;, <,> .

In the syntax < * I > is used to designate an identifier and < * N > is used to designate a number. Further, the special words in the language are capitalized and underlined.

APPENDIX 2: A SAMPLE TRANQUIL PROGRAM

```
BEGIN
REAL SKEWED ARRAY A, B[1:75, 1:75];
INCSET JJ;
MONOSET II(1) [27,6], KK(1) [75, 75];
INTEGER I, J, K;
II ← [2, 10, 13, 15, 21, 24 ];
JJ ← [2, 4, . . ., 74];
FOR (I) SEQ (II) DO
      BEGIN FOR (J) SIM (JJ) DO
                KK ← SET (J:A[I,J] < B[J,I]);
            FOR (K) SIM (KK) DO
                A[I,K] ← A[I+1,K] + B[I,K+1]
      END;
FOR (I,K) SIM (II × KK) DO
            A[I,K] ← A[I+1,K] + B[I,K+1]
END
```

REFERENCES

ABEL, N. E., BUDNIK, P. P., KUCK, D. J., MURAOKA, Y., NORTHCOTE, R. S., and WILHELMSON, R. B. (1969): "TRANQUIL: A Language for an Array Processing Computer", Proc. 1969 Spring Joint Computer Conf., Boston.

BARNES, G. H., BROWN, R. M., KATO, M., KUCK, D. J., SLOTNICK, D. L., STOKES, R. A. (1968): "The ILLIAC IV Computer", IEEE Trans. on Comp., Vol. C-17, p. 746.

BEALS, A. J. (1969): "The Generation of a Deterministic Parsing Algorithm", M. S. Thesis, Report No. 304, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois.

DeREMER, F. L. (1969): "On the Generation of Parsers for BNF Grammars: an Algorithm", Proc. 1969 Spring Joint Comp. Conf., Boston.

FELDMAN, J. A. (1966): "A Formal Semantics for Computer Languages and its Application in a Compiler-Compiler", Comm. Assoc. Comp. Mach., Vol. 9, p. 3.

IVERSON, K. E. (1962): A Programming Language, John Wiley and Sons, Inc., New York, London.

JONES, P. D. (1968): "Operating System Structures", Proc. IFIP Congress, Edinburgh, p. C29.

KUCK, D. J. (1968): "ILLIAC IV Software and Application Programming", IEEE Trans. on Comp. Vol. C-17, p. 758.

LAWRIE, D. H. (1969): "GLYPNIR: A List Processing Lanugage for ILLIAC IV", M. S. Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois.

MARCEAU, I. W., LERMIT, R. J., McMILLAN, J. C., YAMAMOTO, T., and YARDLEY, S. S. (1969): "Linear Programming on ILLIAC IV", Proc. Fourth Australian Comp. Conf., Adelaide.

NORTHCOTE, R. S. (1966): "The Structure and Use of a Compiler-Compiler System", Proc. Third Australian Comp. Conf., Melbourne, p. 339.

TROUT, H. R. G. (1969): "A BNF-like Language for the Description of Syntax Directed Compilers", M. S. Thesis, Report No. 300, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois.

WELLS, M. B. (1964): "Aspects of Language Design for Combinatorial Computing", IEEE Trans. on Comp., Vol. C-13, p. 431.

CR Categories:

4.0, 4.10, 4.12, 4.22, 4.31, 4.41.

Key Words and Phrases:

parallel computation, parallel computer, array computer, operating system, translator writing system, compiler-compiler, syntax, semantics, programming languages, syntax directed compiler, translator, compiler, storage scheme, data structures

FIGURE CAPTIONS

Figure 1. The organization of the ILLIAC IV system.

Figure 2. The structure of the ILLIAC IV operating system, with its modules shown by solid circles.

Figure 3. The structure of the translator writing system.

Figure 4. The partitioning of an array A[1:3, 1:75, 1:75] into blocks for array storage.

Figure 5. Block packing for the array A[1:3, 1:75, 1:75].

UNCLASSIFIED

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)*
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

2a. REPORT SECURITY CLASSIFICATION
UNCLASSIFIED

2b. GROUP

3. REPORT TITLE

SOFTWARE DEVELOPMENT FOR THE ARRAY COMPUTER ILLIAC IV

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*
Research Report

5. AUTHOR(S) *(First name, middle initial, last name)*

Robert S. Northcote

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
March 12, 1969	35	13

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)
46-26-15-305	
b. PROJECT NO. USAF 30(602)4144	DCS Report No. 313
c.	9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*
d.	

10. DISTRIBUTION STATEMENT

Qualified requesters may obtain copies of this report from DCS.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
NONE	Rome Air Development Center Griffiss Air Force Base Rome, New York 13440

13. ABSTRACT

This paper is a survey of the software development for ILLIAC IV. A brief description of the ILLIAC IV hardware system and its memory organization is given. The software is being implemented on a Burroughs B5500 for eventual use on a B6500. The structure of the operating system and the functions of the operating system modules are described to illustrate the somewhat different approach necessary in processing ILLIAC IV programs. The translator writing system, which provides for the automatic generation of syntax directed recognizers and facilitates the construction of other parts of translators, is described. Multipass translators with a simple structure can be obtained from the system on suitable specification of the syntax and semantics of the source language. A brief description of TRANQUIL, an algorithmic language for array processing, is given. Some of the techniques for handling large arrays and mapping functions to specify storage methods for them are mentioned.

DD FORM 1 NOV 65 1473

UNCLASSIFIED

Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
parallel computation						
parallel computer						
array computer						
operating system						
translator writing system						
compiler-compiler						
syntax						
semantics						
programming						
languages						
syntax directed compiler						
translator						
compiler						
storage scheme						
data structures						